|  |  |
| --- | --- |
| Name: | **Subhasish Mukherjee** |
| Lab User ID: | **23SEK3324\_U05** |
| Date: | 09/01/24 |
| Application Name: | [VulnerableJavaWebApplication](https://github.com/SubhAsish1216/VulnerableJavaWebApplication) |

**Follow the below guidelines:**





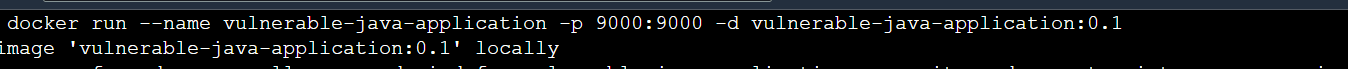
System Architecture:

(Understand the system and document the physical and logical architecture of the system, use the shapes and icons to capture the system architecture)

A screenshot of a computer

Description automatically generated

**Commands:**

****

**Explanation**: In this architecture, an Amazon EC2 (Elastic Compute Cloud) instance serves as the hosting environment for a Java web application running within a container. The EC2 instance is essentially a virtual server in the cloud, providing scalable computing resources. The Java web application, encapsulated in a container, is a lightweight, portable, and isolated unit that ensures consistent deployment across different environments.

The web application exposes its services through an open port, specifically port 9000, allowing external access to the application's functionality. Users can interact with the application by accessing the public IP address associated with the EC2 instance and appending the port number (e.g., http://<public\_ip>:9000) in a web browser. This setup enables remote users to connect to and utilize the web application over the internet.

The EC2 instance facilitates the deployment, management, and scaling of the Java web application, providing the necessary computing resources to ensure optimal performance. This architecture leverages the benefits of cloud computing, such as flexibility and scalability, allowing users to easily access and use the Java web application without the need for physical infrastructure management. The containerization of the application enhances consistency and portability, making it easier to deploy across various environments. Overall, this architecture combines the flexibility of cloud-based infrastructure with containerization to deliver a reliable and accessible Java web application.

Define system’s normal behavior:

(Define the steady state of the system is defined, thereby defining some measurable outputs which can indicate the system’s normal behavior)

This is a basic, self-contained Java web application built with Spring Boot and an embedded H2 database. The application has notable security flaws. It operates over HTTPS on port 9001 and incorporates an embedded H2 database that resets each time the application restarts. This reset feature allows users to revert any unintended changes or security breaches by simply restarting the application. The utilization of Spring Boot ensures a streamlined development and deployment process. However, the mention of security flaws indicates potential vulnerabilities in the application, emphasizing the need for thorough security assessments, updates, and best practices implementation to safeguard against potential exploits. Despite its simplicity, addressing these security concerns is crucial for maintaining the integrity and security of the web application.

Hypothesis:

(During an experiment, we need a hypothesis for comparing to a stable control group, and the same applies here too. If there is a reasonable expectation for a particular action according to which we will change the steady state of a system, then the first thing to do is to fix the system so that we accommodate for the action that will potentially have that effect on the system. For eg: "If one of our database servers fails, our service will automatically switch to a backup server, and users will not experience any downtime or data loss.")



**Known**

If the application experiences any kind of external attack, then its defense mechanisms will effectively mitigate the threat, preventing unauthorized access or data compromise.

If we induce a CPU outage by simulating high load, then the system will automatically scale up to handle increased demand, maintaining performance and availability.

**Unknown**

If a SQL injection attack is executed on the vulnerable Java application, then the application's security measures will may not prevent or mitigate the unauthorized database access attempt. Causing any kind of integrity issues.

Many security vulnerabilities in the Java application may lead to unpredictable consequences, exposing potential weaknesses that could result in unauthorized access, data breaches, or system disruptions. What could happen is not sure and we need to be ready for it.

**Unknown**

**Known**

**Known-Known:** This hypothesis posits that if the system undergoes a CPU outage induced by high load simulation, it will autonomously scale up to meet elevated demand. The expected outcome is to ensure sustained performance and availability, showcasing the system's resilience and capacity to dynamically adapt to resource challenges during increased operational loads.

**Known-Unknown:** This hypothesis suggests that if a SQL injection attack occurs on the vulnerable Java application, the security measures in place may fail to prevent or effectively mitigate the unauthorized database access, potentially resulting in integrity issues. The concern is that the attack might compromise data integrity, emphasizing the importance of robust security measures to safeguard against SQL injection vulnerabilities. It could cause anything.

**Unknown-Unknown:** The Java application's numerous security vulnerabilities pose a significant risk, potentially leading to unforeseen consequences such as unauthorized access, data breaches, or system disruptions. These weaknesses create avenues for exploitation, endangering the confidentiality, integrity, and availability of the system. Mitigating these vulnerabilities requires thorough assessments, proactive security measures, and regular updates to protect against evolving threats. A comprehensive approach, including continuous monitoring and prompt remediation, is crucial to fortify the application's defenses and ensure a robust security posture against the dynamic landscape of potential cyber threats. What could happen is not sure and we need to be ready for it

**Unknown-Known:** If the application encounters any kind of external attack, its defense mechanisms will adeptly counter the threat, thwarting unauthorized access and safeguarding against data compromise. The expectation is that the security measures in place will efficiently identify and neutralize the attack, maintaining the confidentiality and integrity of sensitive information. A successful outcome implies that the application's defenses are robust and responsive, providing a resilient shield against external threats and contributing to a secure operational environment for the protected data and services.

Experiment:

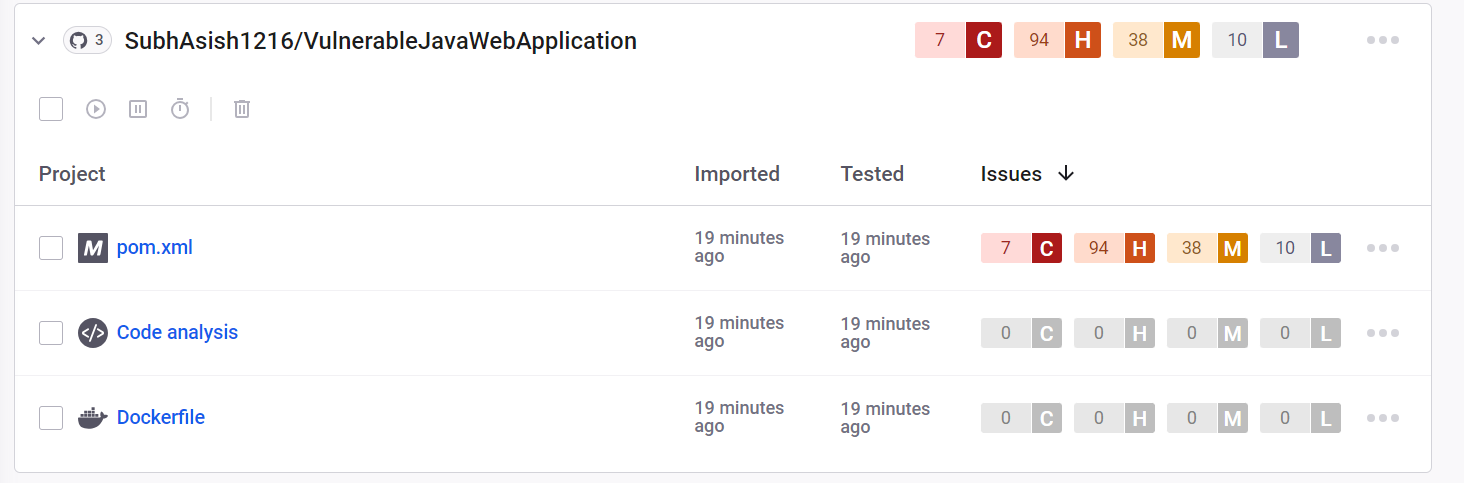
(Document your Preparation, Implementation, Observation and Analysis )

Using some tools for security analysis on this Juice Shop application.

1. **OWASP ZAP**
2. **SNYK**
3. **TRIVY**

**Observation:**

**SNYK: Performing SNYK**



**After performing SNYK there are some critical, high, medium and low vulnerabilities in the pom.xml  
Describing some issues with solution:**

**Spring4Shell, CVE-2022–22965**, is a critical remote code execution vulnerability in the widely used Spring Framework (versions 5.3.0 to 3.5.17, 5.2.0 to 5.2.19, and older unsupported versions). Exploiting this flaw allows attackers to execute code remotely. Affected versions should be patched promptly to mitigate risks.

**Solution:** To address the Spring4Shell (CVE-2022–22965) vulnerability, promptly update the Spring Framework to versions where the issue is patched. Implementing the latest security patches and staying informed about official updates are crucial steps to mitigate the risk of remote code execution.

The vulnerability **(Remote Code Execution) in "com.h2database:h2"** allows attackers to perform code injection, a type of attack enabling malicious code injection through user input fields. Such vulnerabilities are rare but arise when developers dynamically generate code. Mitigation involves reevaluating dynamic code execution, especially with user inputs, to prevent code injection attacks.

**Solution**: To address the Remote Code Execution (RCE) vulnerability in "com.h2database:h2," promptly update the H2 database library to the latest version that includes the necessary security patches. Regularly monitor for updates and follow best practices to minimize the risk of code injection, such as avoiding dynamic code execution with user inputs where possible.

The vulnerability in "**com.fasterxml.jackson.core:jackson-databind**" pertains to insecure deserialization. This occurs when the server deserializes attacker-controlled data, posing a risk of remote code execution. Serialization transforms data for transport, and insecure deserialization can be exploited by attackers to compromise system integrity. Regular updates and secure coding practices are essential for mitigation.

**Solution**: To address the insecure deserialization vulnerability in "com.fasterxml.jackson.core:jackson-databind," update the library to the latest secure version. Additionally, validate and sanitize input data, employ proper authentication, and consider implementing deserialization controls to mitigate the risk of remote code execution through this vector.

The vul**nerability in "org.apache.tomcat.embed:tomcat-embed-core"** exposes a risk of arbitrary code execution, commonly associated with code injection attacks. Code injection occurs when malicious code is inserted through user input, posing a threat to system security. Prevention involves reevaluating dynamic code execution, particularly in scenarios involving user input, and applying secure coding practices. Regularly updating the library is essential for mitigation.

**Solution**: To mitigate the arbitrary code execution vulnerability in "org.apache.tomcat.embed:tomcat-embed-core," promptly update to the latest secure version. Additionally, adopt secure coding practices, validate and sanitize user inputs, and limit dynamic code execution, especially in scenarios where user input is involved, to prevent code injection attacks.

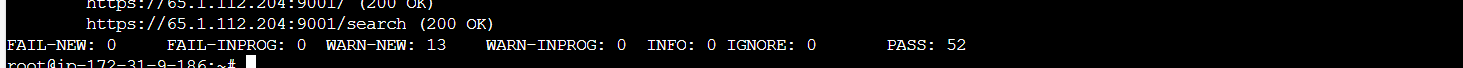
**OWASP ZAP:** Analyzing the application by OWASP ZAP.

**Command:**



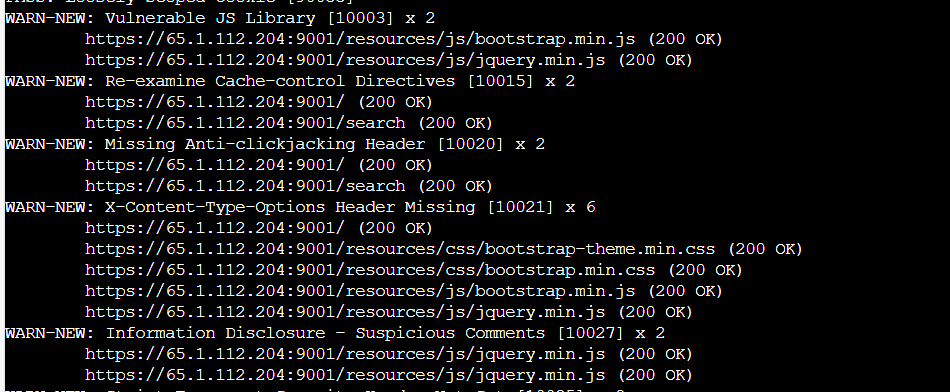
By using this command, we are performing zap base line command on our live application.

**Analysis and result:**



it has passed only 52 tests and 13 are new. It has ignored 0 tests.

By performing this ZAP test, I have got several security issues.



“**Vulnerable JS Library [10003] x 2**" likely refers to a security issue in a JavaScript library with the identifier 10003, repeated twice. This suggests a potential vulnerability or threat associated with the specified library. It is crucial to address such vulnerabilities promptly by updating the library to a patched version or finding an alternative to mitigate potential security risks in web applications.

**Solution:** To address the vulnerability in JS Library [10003] x 2, promptly update to the latest patched version or consider an alternative library without the identified security flaw. Regularly monitor security advisories, apply patches, and follow best practices to ensure a secure web application environment. Additionally, conduct thorough testing after updates to validate the security and functionality of the modified code.

"**Re-examine Cache-control Directives [10015] x 2"** indicates a need to review and assess the Cache-Control directives with the identifier 10015, repeated twice. This suggests a potential issue or misconfiguration related to caching policies in a web application. A careful examination and adjustment of Cache-Control headers are essential to optimize performance, ensure data integrity, and mitigate any caching-related problems in the application.

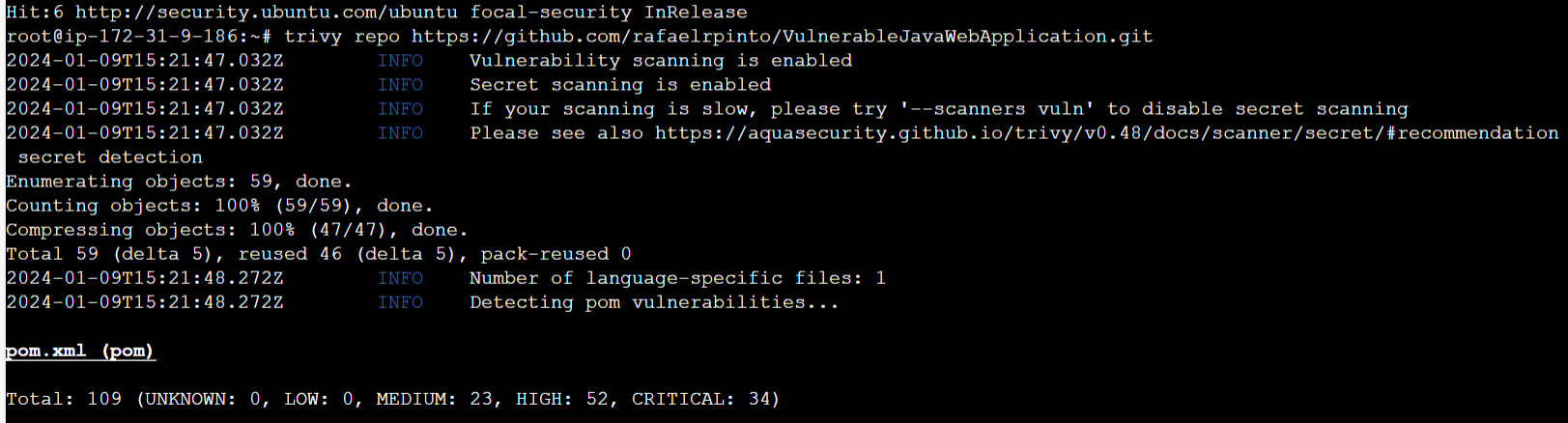
**Solution**: To address the Cache-Control Directives [10015] x 2 issue, carefully review and adjust the caching directives in the web application's headers. Ensure proper configuration to optimize performance while maintaining data integrity. Regularly monitor and update caching settings based on application requirements and security best practices.

**"Missing Anti-clickjacking Header [10020] x 2"** indicates a security concern where the web application lacks a necessary anti-clickjacking header, identified by the code 10020, and this issue is repeated twice. Implementing anti-clickjacking headers, such as X-Frame-Options, is crucial to prevent malicious attacks where a user is tricked into clicking on something different from what they perceive on a website.

**Solution**: To address the "Missing Anti-clickjacking Header [10020] x 2" issue, implement the X-Frame-Options header in the web server configuration. Set it to 'DENY' to prevent any framing of the web content. Regularly update and review security headers to ensure effective protection against clickjacking attacks, enhancing the overall security posture of the web application.

**TRIVY:** Analyzing the repo by using TRIVY

**Command:**



By performing this TRIVY, I have got several security issues.



**Log back versions prior to 1.2.0 contain a serialization vulnerability impacting Socket Server and ServerSocketReceiver** components. Exploiting this flaw, attackers could manipulate serialized data, potentially leading to remote code execution. To mitigate this risk, it is crucial to update Log back to version 1.2.0 or later, implementing the patched release to ensure the security of Socket Server and ServerSocketReceiver components.

**Solution**: To address the Logback serialization vulnerability, update to Logback version 1.2.0 or later, which includes a fix for the issue. Regularly check for security updates, and ensure that your software components are running the latest patched versions. This helps prevent potential remote code execution risks associated with the SocketServer and ServerSocketReceiver components.

**Serializing objects for communication are convenient**, but without proper safeguards, deserialized data can be altered without using accessor functions. Relying on client-side cryptography is risky. Untrusted data may not be well-formed. Allowing unrestricted "gadget chains" during deserialization can enable attackers to execute unauthorized actions. Mitigate risks with secure deserialization practices, input validation, and considering alternative serialization formats like JSON.

**Solution**: To address serialization risks, prioritize secure deserialization practices. Implement input validation, avoiding unrestricted "gadget chains" to prevent unauthorized code execution. Use strong cryptography cautiously, considering its limitations in client-side security. Regularly update libraries and consider alternative serialization formats, such as JSON, to enhance overall application security.

**Faster Jackson-data bind versions 2.8.10 to 2.9.3 have a vulnerability (CVE-2017-7525)** allowing unauthenticated remote code execution. An incomplete fix exposes Object Mapper’s read Value method to malicious JSON input, bypassing an ineffective blacklist if Spring libraries are present. Exploiting this flaw enables attackers to execute code remotely, emphasizing the importance of updating to patched versions and securing class paths to mitigate the risk.

**Solution**: To address the Faster Jackson-data bind vulnerability (CVE-2017-7525), update to versions 2.8.11 or 2.9.4 or later, which contain a complete fix. Additionally, carefully review and manage class paths to minimize the risk of exploitation. Regularly monitor security advisories and promptly apply updates to maintain a secure application environment.

**A deserialization vulnerability in jackson-databind versions prior to 2.8.10 and** 2.9.1 enables unauthenticated users to execute code via malicious input to the readValue method of the ObjectMapper. This issue expands the impact of the previous CVE-2017-7525 flaw by blacklisting additional classes susceptible to exploitation, emphasizing the critical need for users to update to secure versions and mitigate the associated risks.

**Solution**: To address the jackson-databind deserialization flaw, update to version 2.8.10 or 2.9.1 or later, as these contain fixes for the vulnerability. Regularly monitor security advisories, apply patches promptly, and ensure your application doesn't accept untrusted input. Implementing proper input validation and considering alternative serialization formats can further enhance security.